# Reciprocation, square root, inverse square root, and some elementary functions using small multipliers

**Article** *in* IEEE Transactions on Computers · August 2000

DOI: 10.1109/12.863031 · Source: IEEE Xplore

Milos Ercegovac
University of California, Los Angeles
**295** PUBLICATIONS **5,450** CITATIONS
SEE PROFILE

Jean-Michel Muller
Ecole normale supérieure de Lyon
**311** PUBLICATIONS **4,686** CITATIONS
SEE PROFILE

# Reciprocation, Square Root, Inverse Square Root, and Some Elementary Functions Using Small Multipliers

Milos D. Ercegovac, *Member*, *IEEE*, Tomás Lang, *Member*, *IEEE Computer Society*, Jean-Michel Muller, *Member*, *IEEE Computer Society*, and Arnaud Tisserand, *Member*, *IEEE*

**Abstract**—This paper deals with the computation of reciprocals, square roots, inverse square roots, and some elementary functions using small tables, small multipliers, and, for some functions, a final "large" (almost full-length) multiplication. We propose a method, based on argument reduction and series expansion, that allows fast evaluation of these functions in high precision. The strength of this method is that the same scheme allows the computation of all these functions. We estimate the delay, the size/number of tables, and the size/number of multipliers and compare with other related methods.

**Index Terms**—Reciprocal, square root, inverse square root, logarithm, exponential, single-/double-precision operations, small multipliers, Taylor series.

## 1 Introduction

The computation of reciprocal and square root has been considered of importance for many years since these functions appear in many applications. Recently, inverse square root has also received attention because of the increased significance of multimedia and graphics applications. Moreover, because of their similar characteristics, it is considered advantageous to have a single scheme to implement all three functions. We consider such a scheme here. In addition, it allows the computation of logarithms and exponentials.

The progress in VLSI technology now allows the use of large tables with short access time. As a consequence, many methods using tables have emerged during the last decade: high-radix digit-recurrence methods for division and square root [1], [15], inverse square root [16], convergence methods for division and square root [9], combination of table-lookup and polynomial approximation for the elementary functions [8], [6], [12], or (for single precision) use of table-lookups and addition only [14], [3], [10].

We are interested in computations in high precision, such as IEEE-754 double-precision (53-bit significand) format. For double precision, these are hard to achieve with today's technology by direct table lookup, tables and additions, or linear approximations.

The standard scheme to compute reciprocal, square-root, and inverse square root with high precision is based on Newton-Raphson iterations. Although the scheme has a quadratic convergence, the iterations consist of multiplications and additions and are therefore relatively slow. A variation of this method is presented in [5].

We now briefly review other methods. In [2], a method to compute reciprocal, square root, and several elementary functions is presented (and probably could also implement inverse square root). The method is based on series expansion and the implementation consists of several tables, two multipliers, and an adder. For an approximation with relative error $2^{-m}$, the tables have about $m/3$ input bits, which is too large for double precision.

In [13], a method is proposed for double-precision calculations. This requires several tables with an input of 10 bits and rectangular multiplications (typically of $16 \times 56$ bits). In Section 5, we compare this scheme with the one presented here.

The bipartite table methods [10], [3], [11] require the use of tables with approximately $2m/3$ address bits and do not need multiplications to get the result (an addition suffices). These methods might be attractive for single precision calculations, but, with currently available technology, they would require extensively large tables for double precision calculations.

In this paper, we propose a unified algorithm that allows the evaluation of reciprocal, square root, inverse square root, logarithm, and exponential, using one table access, a few "small" multiplications, and at most one "large" multiplication. To approximate a function with about $m$-bit accuracy, we use tables with $m/4$ address bits. This makes our method suitable up to and including double precision.

- *M.D. Ercegovac is with the Computer Science Department, 4731 Boelter Hall, University of California at Los Angeles, Los Angeles, CA 90095. E-mail: milos@cs.ucla.edu.*
- *T. Lang is with Department of Electrical and Computer Engineering, University of California at Irvine, Irvine, CA 92697. E-mail: tlang@ece.uci.edu.*
- *J.-M. Muller is with CNRS-LIP, Ecole Normale Supérieure de Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France. E-mail: jmmuller@ens-lyon.fr.*
- *A. Tisserand is with INRIA-LIP, Ecole Normale Supérieure de Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France. E-mail: Arnaud.Tisserand@ens-lyon.fr.*

*Manuscript received 1 Sept. 1999; revised 1 Feb. 2000; accepted 10 Mar. 2000. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 111784.*

As in other methods of this type, it is possible to obtain an error which is bounded (say by $2^{-m}$). However, it is not possible in general to obtain results that can be directly rounded to nearest. It has been shown [4] that, for the special cases of reciprocal and square root to round to $m$ bits, it is sufficient to compute a result with an error of less than $2^{-2m}$. Similarly, for inverse square root, the error has to be less than $2^{-3m}$. Since this overhead in accuracy might be prohibitive, another alternative is to produce an error of less than $2^{-m-1}$ and determine the exact value by computing the corresponding remainder, which is possible for division and square-root, but not for the transcendental functions [8], [7]. We do not discuss this issue further in this paper and, in the sequel, we will aim to obtain an error which is less than $2^{-m}$, for $m$-bit operands.

## 2 Reciprocal, Square Root, and Inverse Square Root

We want to evaluate reciprocals, square roots, and inverse square roots for operands and results represented by $m$-bit significands. To achieve an error which is smaller than $2^{-m}$, we use an internal datapath of $n$ bits (to be determined) with $n > m$. We do not consider the computation of the exponent since this is straightforward.

Let us call the generic computation $g(Y)$, where $Y$ is the $m$-bit input significand and, as in the IEEE standard, $1 \le Y < 2$. The method is based on the Taylor expansion of the function to compute, which converges with few terms if the argument is close to 1. Consequently, the method consists of the following three steps:

1. **Reduction.** From $Y$, we deduce an $n$-bit number $A$ such that $-2^{-k} < A < 2^{-k}$. To produce a simple implementation that achieves the required precision, we use $k = n/4$. For the functions considered, we obtain $A$ as
$$A = Y \times \hat{R} - 1, \tag{1}$$
where $\hat{R}$ is a $(k+1)$-bit approximation of $1/Y$. Specifically, define $Y^{(k)}$ as $Y$ truncated to the $k$th bit. Then,
$$Y^{(k)} \le Y < Y^{(k)} + 2^{-k}.$$
Hence,
$$1 \le \frac{Y}{Y^{(k)}} < 1 + 2^{-k}. \tag{2}$$
Using one lookup in a $k$-bit address table, one can find $\hat{R}$ defined as $1/Y^{(k)}$ rounded *down* (i.e., truncated) to $k+1$ bits. Then,
$$-2^{-k-1} < \hat{R} - \frac{1}{Y^{(k)}} \le 0.$$
Therefore, since $1 \le Y^{(k)} < 2$,
$$1 - 2^{-k} < \hat{R}Y^{(k)} \le 1. \tag{3}$$
From (2) and (3), we get
$$1 - 2^{-k} < \hat{R}Y < 1 + 2^{-k}. \tag{4}$$
The *reduced argument* $A$ is such that $g(Y)$ can be easily obtained from a value $f(A)$, which is computed during the next step.
2. **Evaluation.** We compute an approximation of $B = f(A)$ using the series expansion of $f$, as described below.
3. **Postprocessing.** This is required because of the reduction step. Since reduction is performed by multiplication by $\hat{R}$, we obtain $g(Y)$ from $B = f(A)$ as
$$g(Y) = M \times B,$$
where $M = h(\hat{R})$. The value of $M$ depends on the function and is obtained by a similar method as $\hat{R}$. Specifically,
   - for reciprocal $M = \hat{R}$,
   - for square root $M = 1/\sqrt{\hat{R}}$,
   - for inverse square root $M = \sqrt{\hat{R}}$.

Hence, although $\hat{R}$ is the same for all functions considered here, $M$ depends on the function being computed. There is a different table for $M$ for each function we wish to implement. Let us now consider the evaluation step.

### 2.1 Evaluation Step

In the following, we assume that we want to evaluate $B = f(A)$, with $|A| < 2^{-k}$. The Taylor series expansion of $f$ is
$$f(A) = C_0 + C_1A + C_2A^2 + C_3A^3 + C_4A^4 + \dots \tag{5}$$
at the origin where the $C_i$s are bounded.

Since $-2^{-k} < A < 2^{-k}$, $A$ has the form
$$A = A_2z^2 + A_3z^3 + A_4z^4 + \dots, \tag{6}$$
where $z = 2^{-k}$ and $|A_i| \le 2^k - 1$.

Our goal is to compute an approximation of $f(A)$, correct to approximately $n = 4k$ bits, using small multiplications. From the series (5) and the decomposition (6), we deduce
$$\begin{aligned} f(A) = C_0 &+ C_1\left(A_2z^2 + A_3z^3 + A_4z^4\right) \\ &+ C_2\left(A_2z^2 + A_3z^3 + A_4z^4\right)^2 \\ &+ C_3\left(A_2z^2 + A_3z^3 + A_4z^4\right)^3 \\ &+ C_4\left(A_2z^2 + A_3z^3 + A_4z^4\right)^4 + \dots . \end{aligned} \tag{7}$$

After having expanded this series and dropped out all the terms of the form $W \times z^j$ that are less than or equal to $2^{-4k}$, we get (see the Appendix)
$$f(A) \approx C_0 + C_1A + C_2A_2^2z^4 + 2C_2A_2A_3z^5 + C_3A_2^3z^6. \tag{8}$$
We use this last expression to approximate reciprocals, square roots, and inverse square roots. In practice, when computing (8), we make another approximation: after having computed $A_2^2$, obtaining $A_2^3$ would require a $2k \times k$ multiplication. Instead of this, we take only the $k$ most-significant bits of $A_2^2$ and multiply them by $A_2$.

Now, we determine the coefficients for the three functions

- For reciprocal, $|C_i| = 1$ for any $i$, and

$$\begin{aligned}\frac{1}{1+A} &\approx 1 - A_2z^2 - A_3z^3 + \left(-A_4 + A_2^2\right)z^4 \\ &\quad + 2A_2A_3z^5 - A_2^3z^6 \\ &\approx (1-A) + A_2^2z^4 + 2A_2A_3z^5 - A_2^3z^6.\end{aligned} \tag{9}$$

- For square root, $C_0 = 1$, $C_1 = 1/2$, $C_2 = -1/8$, $C_3 = 1/16$. This gives

$$\sqrt{1+A} \approx 1 + \frac{A}{2} - \frac{1}{8}A_2^2z^4 - \frac{1}{4}A_2A_3z^5 + \frac{1}{16}A_2^3z^6. \tag{10}$$

- For inverse square root, $C_0 = 1$, $C_1 = -1/2$, $C_2 = 3/8$, $C_3 = -5/16$. This gives

$$1/\sqrt{1+A} \approx 1 - \frac{A}{2} + \frac{3}{8}A_2^2z^4 + \frac{3}{4}A_2A_3z^5 - \frac{5}{16}A_2^3z^6. \tag{11}$$

## 2.2 Error in the Evaluation Step

We now consider the error produced by the evaluation step described above. In the Appendix, we prove the following result:

**Theorem 1.** *f(A) can be approximated by*

$$C_0 + C_1A + C_2A_2^2z^4 + 2C_2A_2A_3z^5 + C_3A_2^3z^6$$

*(where we use the k most-significant bits*[1] *only of* $A_2^2$ *when computing* $A_2^3$*), with an error less than*

$$2^{-4k}\left(\frac{C_{max}}{1-2^{-k}} + 3|C_2| + 4|C_3| + 8.5\max\{|C_2|,|C_3|\} \times 2^{-k}\right)$$

*with* $C_{max} = \max_{i\geq 4}|C_i|$*, and* $k \geq 5$.

*In particular, assuming* $|C_i| \leq 1$ *for any i (which is satisfied for the functions considered in this paper), this error is less than*

$$\epsilon = 2^{-4k}(1.04C_{max} + 3|C_2| + 4|C_3| + 0.27).$$

Now, we determine the error bound $\epsilon$ for the three functions, *assuming A is exactly equal to*

$$A = A_2z^2 + A_3z^3 + A_4z^4.$$

- For reciprocal, since $|C_i| = 1$ for all $i$,

$$\epsilon = 8.31 \times 2^{-4k}.$$

- For square root, $C_2 = 2^{-3}$, $C_3 = 2^{-4}$, and $C_{max} = 5 \times 2^{-7}$,

$$\epsilon = 0.94 \times 2^{-4k}.$$

- For inverse square root, $C_2 = 3 \times 2^{-3}$, $C_3 = -5 \times 2^{-4}$, and $C_{max} = 35 \times 2^{-7}$,

$$\epsilon = 2.93 \times 2^{-4k}.$$

These errors are committed by evaluating (8) in infinite precision arithmetic (and using the $k$ most-significant bits of $A_2^2$ only). To this, we have to add the following two errors:

- $A$ has more than $4k$ bits. Consequently, we have to add the error $2^{-4k-1}\max_A f'(A)$ due to having rounded $A$ to $A_2z^2 + A_3z^3 + A_4z^4$.
- If the evaluation step returns a value rounded to the nearest multiple of $2^{-4k}$, we have to add the maximum error value $2^{-4k-1}$ due to this rounding.

All this gives an upper bound $\epsilon_{\text{eval}}$ due to the evaluation step.

TABLE 1
Upper Bound on the Total Absolute Error

| Operation | $M_{\max}$ | Total Error |
|---|---|---|
| Reciprocal | 1 | $9.31 \times 2^{-4k}$ |
| Square root | $\sqrt{2}$ | $2.39 \times 2^{-4k}$ |
| Inv. square root | 1 | $3.68 \times 2^{-4k}$ |

## 2.3 Total Error and Value of $k$

We now take into account the postprocessing step (multiplication by $M$). To get an upper bound $\epsilon_{\text{total}}$ on the total computation error, we multiply $\epsilon_{\text{eval}}$ by the maximum possible value of $M$. We do not include an error due to rounding $M$ to $n$ bits: It is preferable to round $M$ to $m$ bits directly. Table 1 gives the value of $\epsilon_{\text{total}}$. If a $(3k+1) \times (3k+2)$ multiplier is used for the postprocessing step (as suggested in Section 3.3), then we need to add $0.5 \times 2^{-4k}$ to this value.

Now, let us determine the value of $k$. Since the computed final result $g(Y)$ is between $1/2$ and $1$ for reciprocation, between 1 and $\sqrt{2}$ for square root, and between $1/\sqrt{2}$ and 1 for inverse square-root, the first nonzero bit of the result is of weight $2^0$ for square-root and of weight $2^{-1}$ for the other two functions. Considering the error given in Table 1, the required values of $n$ and $k$ are given in Table 2.

## 3 Implementation

We now describe implementation aspects of the proposed method. Fig. 1 shows a functional representation of the general architecture. In the sequel, we assume that $A$ is in the sign-magnitude form which requires complementation. The multiplications produce products in the signed-digit form, and the addition of the four terms in the evaluation

TABLE 2
Values of $k$ for Functions Evaluated ($n = 4k$)

| Format | Reciprocal | Square root | Inverse square root |
|---|---|---|---|
| SP ($m = 24$) | 7 | 7 | 7 |
| DP ($m = 53$) | 15 | 14 | 14 |

SP—single precision with faithful rounding;
DP—double precision with faithful rounding.

1. It would be more accurate to say *digits* since it is likely that, in a practical implementation, $A_2^2$ will be represented in a redundant (e.g., carry-save or borrow-save) representation.

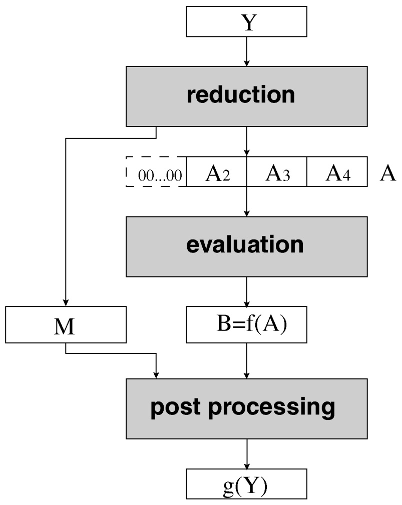

Fig. 1. General organization.

step is performed using signed-digit adders. Modifications for using different number representations are straightforward.

### 3.1 Reduction

Fig. 2 shows a functional representation of the reduction module which computes $A$ and $M$ from $Y$. The factor $M$ is obtained by table lookup from block `ctb` for functions other than reciprocal.

### 3.2 Evaluation

The evaluation step computes (9), (10), and (11). All three require the computation of $A_2^2$, $A_2A_3$, and $A_2^3$. As indicated before, for $A_2^3$ we use the approximation

$$A_2^3 \approx (A_2^2)_{high} \times A_2.$$

Consequently, these terms can be computed by three $k$ by $k$ multiplications. Moreover, the first two can be performed in parallel. Alternatively, it is possible to compute the terms by two multiplications as follows:

$$B_1 = A_2^2 + 2A_2A_3z = A_2 \times (A_2 + 2A_3z)$$
$$\text{and } A_2^3 \approx (B_1)_{high} \times A_2.$$

The first of the two multiplications is of $k \times 2k$ bits and the second is of $k \times k$.

Then, the terms (either the output of the three multiplications or of the two multiplications) are multiplied by the corresponding factors which depend on the function, as shown in Table 3. Note that, for division and square root, these factors correspond to alignments, whereas, for inverse square root, multiplications by 3 and 5 are required.[2] Finally, the resulting terms are added to produce $B$.

Fig. 3 shows the weights of these terms in the case of the reciprocal function. The sum of these terms is rounded to the nearest multiple of $2^{-4k}$. As shown in Fig. 3, this gives a $(3k+1)$-bit number $\hat{B}$. Then, $B$ is equal to $1+\hat{B}$. An implementation is shown in Fig. 4.

### 3.3 Postprocessing

The postprocessing (Fig. 5) consists in multiplying $B$ by $M$, where $M = h(\hat{R})$ depends on the function and is computed during the reduction step. Since $B = 1 + \hat{B}$ and $|\hat{B}| < 2^{-k+1}$, to use a smaller multiplier it is better to compute

$$g(Y) = M \times B = M + M \times \hat{B}. \quad (12)$$

TABLE 3
Multiplication Factors

| Function | $B_1$ | $A_2^3$ |
|---|---|---|
| Reciprocal | 1 | 1 |
| Square root | -1/8 | 1/16 |
| Inverse Square root | 3/8 | -5/16 |

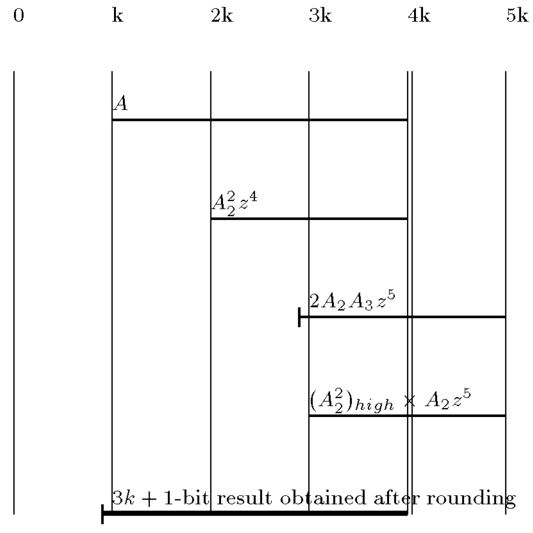
0
k
2k
3k
4k
5k
A
$A_2^2z^4$
$2A_2A_3z^5$
$(A_2^2)_{high} \times A_2z^5$
$3k+1$-bit result obtained after rounding

Fig. 3. The terms added during the evaluation step for reciprocal.

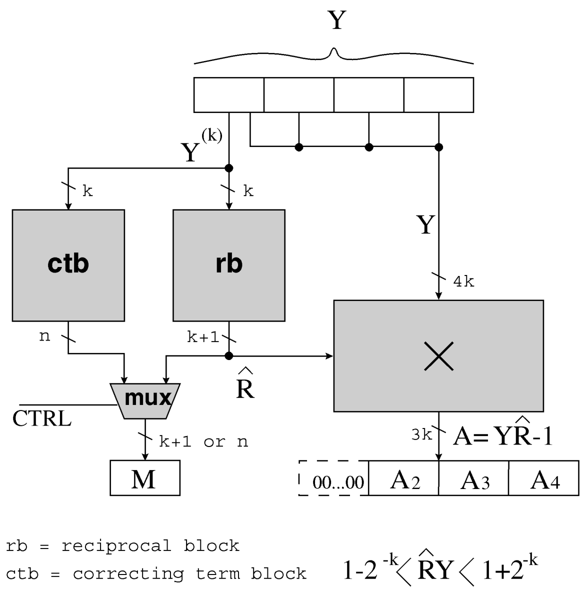

Fig. 2. Organization of the reduction module. $M$ depends on the function being computed.

2. These "multiplications" will be implemented as (possibly redundant) additions since $3 = 2 + 1$ and $5 = 4 + 1$.

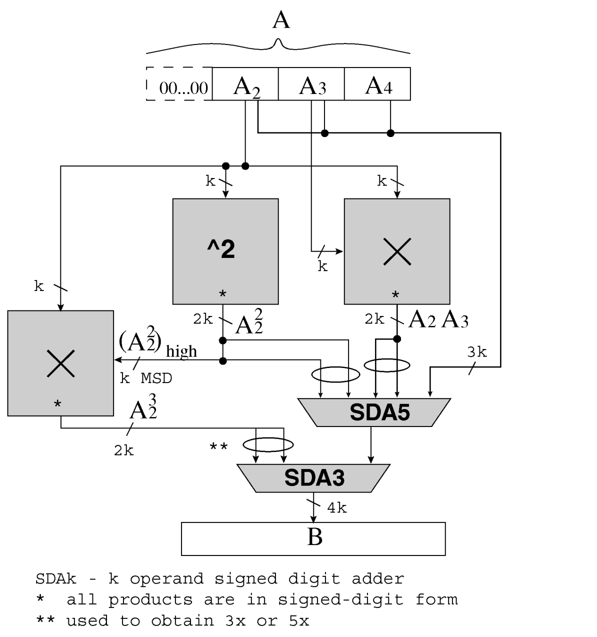
A
00...00
A2
A3
A4
k
k
k
k
^2
k
2k
A2^2
2k
A2 A3
(A2^2) high
k MSD
3k
A2^3
SDA5
2k
**
SDA3
4k
B
SDAk - k operand signed digit adder
* all products are in signed-digit form
** used to obtain 3x or 5x

Fig. 4. Functional representation of the evaluation module. (The computations of $A_2^2$ and $A_2A_3$ can be regrouped into one rectangular multiplication.)

Note also that, for square root and inverse square root, in the multiplication it suffices to use the bits of $M$ of weight larger than or equal to $2^{-3k-t}$, where $t$ is a small integer. Since the error due to this truncation is smaller than or equal to $2^{-4k+1-t}$, choosing $t = 2$ makes the error bounded by $0.5 \times 2^{-4k}$ and allows the use of a $(3k + 1) \times (3k + 2)$-bit multiplier. Hence, although we need to store $n$ bits of $M$ (to be added in (12)), only $3k + 2$ bits will be used in the multiplication $M \times \hat{B}$.

Table 4 shows the operation that must be performed during the postprocessing step and the value of $M$ that must be used.

## 4 Estimation of Execution Time and Hardware

We now evaluate the method proposed in terms of execution time and hardware required. This evaluation serves for the comparisons presented in the next section.

### 4.1 Execution Time

The critical path is given by the following expression:

$$T_{crit} = t_{rb} + tm_{3k\times k} + 2tm_{k\times k} + ta_{4k} + tm_{3k\times 3k},$$

where $t_{rb}$ is the table access time, $tm$ multiplication, and $ta$ addition time.

For instance, for double precision with faithful rounding and implementation of $rb$ directly by table, we obtain the sum of the following delays:

- Access to table of 15 or 14 input bits.
- One multiplication of $46 \times 16$ bits (with product in conventional form).
- Two multiplications of $15 \times 15$ bits (with product in redundant form).
- Signed-digit addition of the four terms. This addition is the most complex for inverse square root (10): All three multiplications produce signed-digit results and, because the coefficients 3/8, 3/4, and -5/16 are replaced by shifts and adds, this leads to $3 \times 2$ signed-digit operands with a total of $1 + 6 = 7$. The multiplications $\frac{3}{8}A_2^2$ and $\frac{3}{4}A_2A_3$ are performed in parallel, followed by a 5-to-1 signed-digit addition SDA5. This addition is performed concurrently with the multiplication $\frac{5}{16}A_2^3$, which produces two operands, so a 3-to-1 signed-digit addition SDA3 completes the critical path in the evaluation module. The result of this addition is used as the multiplier of the next multiplication; consequently, it is possible to directly recode the signed-digit form to radix-4 multiplier.
- Multiplication of $45 \times 43$ bits (with product in conventional form).

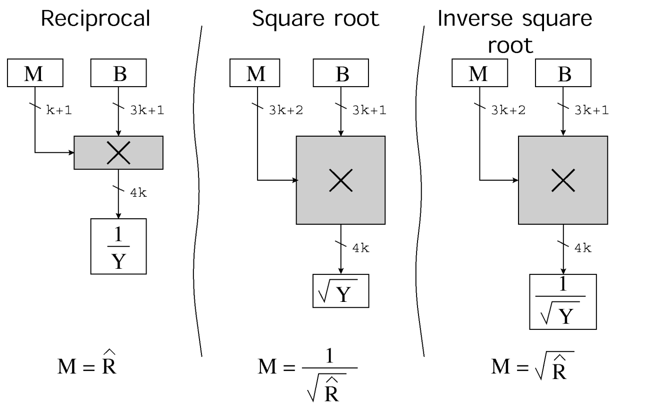

Fig. 5. Functional representation of the postprocessing module.

### 4.2 Hardware

Table 5 gives the table sizes and number of various operations required by our method, depending on the function being computed, and the value of $k$. Table 6 gives the required table sizes depending on the function computed and the format (single precision and double precision).

## 5 Comparison with Other Methods

We restrict our comparison to three methods which also deal with reciprocals, square roots, and inverse square roots. We briefly review these methods and then compare

TABLE 4
Operation Performed during the Postprocessing Step

| Function | Value of $M$ | Size of $M$ | Operation |
|---|---|---|---|
| $1/Y$ | $\hat{R}$ | $k+1$ bits | $M \times B$ |
| $\sqrt{Y}$ | $1/\sqrt{\hat{R}}$ | $n$ bits | $M \times B$ |
| $1/\sqrt{Y}$ | $\sqrt{\hat{R}}$ | $n$ bits | $M \times B$ |

TABLE 5
Table Sizes and Number of Operations for Our Method

| | |
|---|---|
| **Reciprocal** | |
| Table size [bits] | $(k+1) \times 2^k$ |
| Small/large multipliers | 5 (2*) / 0 |
| **Square and inverse square root** | |
| Table size [bits] | $(k+1+n) \times 2^k$ |
| Small/large multipliers | 4 (2*) / 1 |

"Small": $k \times n$ or $k \times k$ multiplication;
"large": $(3k+1) \times (3k+2)$ multiplication; * - in parallel.

estimates of latency (delay) and of cost (mainly of multipliers and tables) for 53-bit precision.

## 5.1 Newton-Raphson Iteration

The well-known Newton-Raphson (NR) iteration for reciprocal

$$x_{i+1} = x_i + x_i(1 - Yx_i) \tag{13}$$

converges quadratically to $1/Y$ provided that $x_0$ is close enough to $1/Y$. We use a $k$-bit address table to obtain $x_0$ and perform the intermediate calculations using an $n$-bit arithmetic. To compare with our method, we assume $n \approx 4k$. The first approximation $x_0$ is the number $\hat{Y}$ of Section 2, a $k$-bit approximation of $1/Y$. To get $x_1$, two $k \times n$-bit multiplications are required. Since $x_1$ is a $2k$-bit approximation of $1/Y$, it suffices to use its $2k$ most-significant bits to perform the next iteration. After this, one needs to perform two $2k \times n$-bit multiplications to get $x_2$, which is an $n$-bit approximation of $1/Y$. For $k = 15$, the NR method requires:

- one lookup in a 15-bit address table;
- two $15 \times 30$-bit multiplications ($Y$ truncated to 30 bits);
- two $30 \times 60$-bit multiplications.

The multiplications that occur cannot be performed in parallel.

The NR iteration for reciprocal square-root

$$x_{i+1} = \frac{1}{2} x_i \left(3 - Yx_i^2\right) \tag{14}$$

has convergence properties similar to those of the NR iteration for reciprocal. Assuming (as previously) that we use a $k$-bit address table and that we perform the intermediate calculations using an $n$-bit arithmetic, with $k = 14$ and $n = 56$ (which are the values required for faithfully rounded double precision square root or inverse square root), computing an inverse square-root using the NR iteration requires:

- one lookup in a 14-bit address table;
- three $14 \times 56$-bit multiplications;
- three $28 \times 56$-bit multiplications.

In the implementation, we assume using a shared $30 \times 60$ multiplier.

Computing a square-root requires the same number of operations plus a final "large" ($56 \times 56$-bit) multiplication.

TABLE 6
Tables Required by Our Method (in Bytes)

| Format | Reciprocal | Square root | Inverse square root | All 3 functions |
|---|---|---|---|---|
| SP | 128 | 576 | 576 | 1024 |
| DP | 65K | 142K | 142K | 289K |

SP - single precision ($m = 24$)with faithful rounding;
DP - double precision ($m = 53$)with faithful rounding.

## 5.2 Wong and Goto's Method

The method presented by Wong and Goto in [14] requires tables with $m/2$ address bits, where $m$ is the number of bits of the significand of the floating-point arithmetic being used. This makes that method inconvenient for double-precision calculations. In [13], they suggest another method using table-lookups and rectangular multipliers.

The method for computing reciprocals is as follows: Let us start from the input value $Y = 1.y_1y_2 \dots y_{53}$. The first 10 bits of $Y$ are used as address bits to get from a table

$$r_0 = \left\lfloor \frac{1}{1.y_1y_2 \dots y_{10}} \right\rfloor.$$

Then, compute $r_0 \times Y$ using a rectangular multiplier. The result is a number $A$ of the form:

$$A = 1 - 0.000 \dots 0a_9a_{10} \dots a_{18} \dots a_{56}.$$

Then, using a rectangular multiplier, compute:

$$\begin{aligned} B &= A \times (1 + 0.000 \dots 0a_9a_{10} \dots a_{18}) \\ &= 1 - 0.000000 \dots 00b_{17}b_{18} \dots b_{26} \dots b_{56}. \end{aligned}$$

Again, using a rectangular multiplier, compute:

$$\begin{aligned} C &= B \times (1 + 0.000000 \dots 00b_{17}b_{18} \dots b_{26}) \\ &= 1 - 0.00000000000 \dots 0000c_{25}c_{26} \dots c_{56}. \end{aligned}$$

In parallel, use the bits $b_{27}b_{28} \dots b_{35}$ as address to get from a table $\beta$ consisting of the nine most significant bits of $(0.0000 \dots b_{27}b_{28} \dots b_{35})^2$. The final result is:

$$\begin{aligned} \frac{1}{Y} \approx r_0 &\times 1.00000 \dots a_9a_{10} \dots a_{18} \\ &\times 1.000000 \dots 00b_{17}b_{18} \dots b_{26} \\ &\times (1.000000000 \dots 000c_{25}c_{26} \dots c_{56} + \beta). \end{aligned} \tag{15}$$

Fig. 6 illustrates the computational graph for 56-bit reciprocal computation.

Therefore, this method for reciprocation requires one table look-up in a 10-bit address table, one look-up in a 9-bit address table, five rectangular $10 \times 56$ multiplications, and one $56 \times 56$ multiplication. The critical path is roughly

$$t_{WG} = t_{LUT10} + 3 \times t_{MULT(10\times56)} + t_{MULT(56\times56)}. \tag{16}$$

To compute reciprocal square-roots, the Wong-Goto method uses one look-up in an 11-bit address table, one look-up in a 9-bit address table, nine rectangular multiplications, and one full multiplication. The critical path

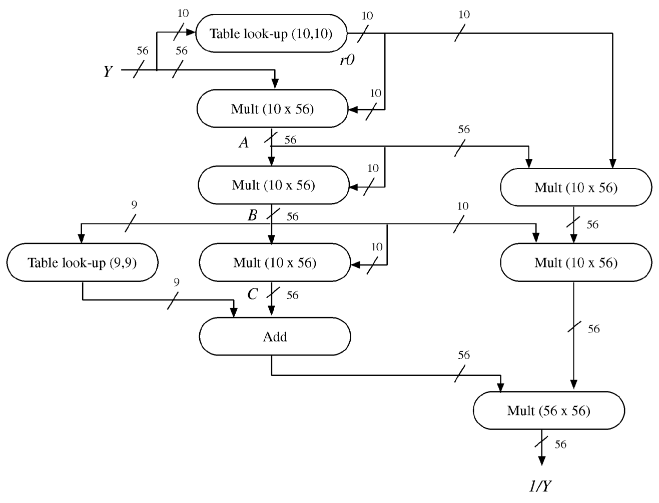

Fig. 6. The Wong-Goto reciprocal computation.

consists of one 11-bit table lookup, five rectangular multiplications, and one full multiplication.

## 5.3 Method Proposed by Ito, Takagi, and Yajima

This proposal [5] considers the operations division (actually reciprocal) and square root performed using a multiply-accumulate unit (as in Newton-Raphson's method). It discusses a linear initial approximation and proposes accelerated convergence methods. Since the linear initial approximation can also be used for the method proposed in this paper, for comparison purposes we do not include it and use the direct table lookup approach.

For reciprocal, the proposal in [5] results in a speedup with respect to traditional NR by modifying the recurrence so that additional accuracy is achieved by each multiply-accumulate and by adding a cubic term, which requires an additional table look-up. For square root, a novel direct algorithm is proposed, instead of going through inverse square root, as done in NR.

For the comparison of reciprocal, we use a look-up table of 15 input bits for the initial approximation (direct table method). In this case, three multiply-accumulate operations are required and the cubic term uses a table of nine input bits.

For square root, also with a table of 15 input bits for the initial approximation, four multiply-accumulate operations are required.

## 5.4 Estimates of Delay and Cost

We now estimate the delays and costs (size of tables and size of multipliers) of the schemes. Following [14], the delays are expressed in terms of $\tau$—the delay of a complex gate, such as one full adder. In this unit, we estimate the delays of multipliers and tables; these delays can vary somewhat depending on the technology and implementation, but, since all schemes use the same modules, the relative values should not vary significantly.

The delay on the critical path of a multiplier is the sum of 1, 2, and 3:

1. Radix-4 multiplier recoding, multiple generation and buffering: $2\tau$.

TABLE 7
Delays of Multipliers: $t_{recode} + t_{reduce} + t_{CPA}$ (in $\tau$s)

| Size | Delay [$\tau$] |
|---|---|
| $10 \times 56$ | 2+3+4=9 |
| $15 \times 15$ | 2+4+3=9 |
| $15 \times 30$ | 2+4+4=10 |
| $16 \times 56$ | 2+4+4=10 |
| $30(28) \times 60(56)$ | 2+5+4=11 |
| $45 \times 45$ | 2+6+4=12 |
| $56 \times 56$ | 2+6+4=12 |

2. Partial product reduction array: $1\tau \times$ number of SDA (Signed-Digit Adder) stages;
3. Final CPA (Carry Propagate Adder)—when needed: $4\tau$ for $> 30$ bits; $3\tau$ for $\leq 30$ bits.

The delays of the various multipliers used are summarized in Table 7.

For the delay of a look-up table with 14-15 address bits, we estimate around $8\tau$ and with 10-12 bits around $5\tau$. The size is given directly in Table 8.

From the description of the methods given above, we summarize their characteritics in Table 8 and obtain the estimates of Table 9. We conclude that, for reciprocal, our method has a similar delay as the other schemes, but is significantly faster for square root and for inverse square root. On the other hand, the Wong-Goto method requires smaller tables.

TABLE 8
Modules Required for Double Precision Computation of Reciprocal, Square Root, and Inverse Square Root in Related Methods

| | |
|---|---|
| **NR** | |
| Table delay [$\tau$]: 8 | Multipliers: |
| Table 1: $2^{15} \times 15$ | $15 \times 30$ |
| Table 2: $2^{14} \times 14$ | $14 \times 56$ |
| | $30 \times 60$ |
| (for square root) | $56 \times 56$ |
| **Wong and Goto** | |
| Table delay [$\tau$]: 5 | Multipliers: |
| Table 1: $2^{10} \times 10$ | $10 \times 56$ (2) |
| Table 2: $2^{9} \times 9$ | $56 \times 56$ |
| Table 3: $2^{11} \times 22$ | |
| Table 4: $2^{11} \times 11$ | |
| Table 5: $2^{9} \times 9$ | |
| **Ito, Takagi and Yajima** | |
| Table delay [$\tau$]: 8 | Multipliers: |
| Table 1: $2^{15} \times 15$ | $56 \times 56$ |
| Table 2: $2^{9} \times 9$ | |
| Table 3: $2^{10} \times 119$ | |
| Table 4: $2^{8} \times 8$ | |
| Table 5: $2^{14} \times 14$ | |

TABLE 9
Estimation of Total Delays (in $\tau$s) for Double Precision Computation of Reciprocals, Square Roots, and Inverse Square Roots

| Method | Reciprocal | Square root | Inverse square root |
|---|---|---|---|
| **NR** | 58 | 87 | 75 |
| **Wong and Goto** | 48 | 78 | 66 |
| **Ito, Takagi and Yajima** | 47 | 60 | 75* |
| **Our** | 48 | 48 | 48 |

$*$ assumed to be implemented as in NR method; $\tau$ - the delay of complex gate.

| | |
|---|---|
| **NR** | |
| Recip. : | 8 (table) + 2x10 (15x30 mults) + 4 (add) + 2x11 (30x60 mults) + 4 (add) = 58 |
| Inv. sqrt: | 8 (table) + 3x10 (14x56 mults) + 2 (sub) + 3x11 (28x56 mults) + 2 (sub) = 75 |
| Sqrt: | Inv. SQRT + 12 (56x56 mult) = 87 |
| **Wong and Goto** | |
| Recip. : | 5 (table) + 3x9 (10x56 mults) + 4 (add) + 12 (56x56 mult) = 48 |
| Inv. sqrt: | 5 (table) + 5x9 (10x56 mults) + 4 (add) + 12 (56x56 mult) = 66 |
| Sqrt: | Inv. SQRT + 12 (56x56 mult) = 78 |
| **Ito, Takagi and Yajima** | |
| Recip: | 8 (table) + 3x12 ( 56x56 mult) + 3 (acc) = 47 |
| Sqrt: | 8 (table) + 4x12 (56x56 mult) + 4 (acc) = 60 |
| **Our** | |
| Recip: | 8 (table) + 10 (16x56 mult) [a] + 2x6 (15x15 mult with SD product) + 4 (SDA) [b] + 4 (CPA) + 10 (16x46 mult) [c] = 48 |
| Sqrt/ Inv. Sqrt | 18 (reduction) + 16 (evaluation) + 2 (recode) + 12 (44 x 43 mult) = 48 |

[a] reduction; [b] evaluation; [c] post-processing.

## 6 Elementary Functions

Using the same basic scheme, our method also allows computation of some of the elementary functions. We briefly describe this below. Implementation is not discussed: It is very similar to what we have previously described for reciprocal, square root, and inverse square root.

### 6.1 Computation of Logarithms

In a similar fashion, we get:

$$\ln(1+A) \approx A - \frac{1}{2}A_2^2 z^4 - A_2 A_3 z^5 + \frac{1}{3}A_2^3 z^6.$$

Again, we only need to compute $A_2^2$, $A_2A_3$, and $A_2^3$. The multiplication by $\frac{1}{3}$ can be done with a small multiplier. The postprocessing step is performed as $g(Y) = M + B$, where $M = -ln(\hat{R})$ and $B = ln(1 + A)$. When the argument is close to 1, no reduction is performed and, consequently, there is no cancellation.

### 6.2 Computation of Exponentials

Now, let us assume that we want to evaluate the exponential of an $n$-bit number $Y = 1 + A_1z + A_2z^2 + A_3z^3 + A_4z^4$, where $z = 2^{-k}$ ($k = n/4$), and the $A_i$s are $k$-bit integers. We suggest first computing the exponential of

$$A = A_2z^2 + A_3z^3 + A_4z^4,$$

using a Taylor expansion, and then to multiply it by the number

$$M = \exp(1 + A_1 z).$$

$M$ will be obtained by looking up in a $k$-bit address table.

The exponential of $A$ can be approximated by:

$$1 + A + \frac{1}{2}A_2^2 z^4 + A_2A_3z^5 + \frac{1}{6}A_2^3 z^6. \tag{17}$$

This shows that the same architecture suggested in Section 3 can be used as well for computing exponentials, with similar delay and accuracy.

TABLE 10
Summary of the Proposed Method

| | |
|---|---|
| **Reciprocal** | |
| Table size [bits] | $(k+1) \times 2^k$ |
| Small/large multipliers | 5 (2*) / 0 |
| **Square/inverse square root** | |
| Table size [bits] | $(k+1+n) \times 2^k$ |
| Small/large multipliers | 4 (2*) / 1 |
| **Logarithm** | |
| Table size [bits] | $(k+1+n) \times 2^k$ |
| Small/large multipliers | 4 (2*) / 0 |
| **Exponential** | |
| Table size [bits] | $n \times 2^k$ |
| Small/large multipliers | 3 (2*) / 0 |
| **Combined (all five)** | |
| Table size [bits] | $(k+1+4n) \times 2^k$ |
| Small/large multipliers | 5 (2*) / 1 |

"Small": $k \times n$ or $k \times k$ multiplication; "large": $(3k+1) \times (3k+2)$ multiplication; * - in parallel $k = 7$ for single precision ($m = 24$); $k = 15(14)$ for double precision ($m = 53$); $n = 4k$.

## 7 Conclusion

We have proposed a method for computation of reciprocals, square-roots, inverse square-roots, logarithms, and exponentials. Table 10 summarizes the key implementation requirements in evaluating these functions. The strength of our method is that the same basic computations are performed for all these various functions. As shown in the section on comparisons, in double precision for reciprocal our method requires a computational delay quite close to other related methods, but it is significantly faster for square root and for inverse square root. We have considered only faithful rounding.

## Appendix

To prove the theorem, let us start from the series (7):

$$\begin{aligned} f(A) = C_0 &+ C_1\left(A_2z^2 + A_3z^3 + A_4z^4\right) \\ &+ C_2\left(A_2z^2 + A_3z^3 + A_4z^4\right)^2 \\ &+ C_3\left(A_2z^2 + A_3z^3 + A_4z^4\right)^3 \\ &+ C_4\left(A_2z^2 + A_3z^3 + A_4z^4\right)^4 + \dots . \end{aligned} \tag{18}$$

Let us keep in mind that $A = A_2z^2 + A_3z^3 + A_4z^4$ is obviously less than $2^{-k}$. If we drop out from the previous series the terms with coefficients $C_4, C_5, C_6, C_7, \ldots$, the error will be:

$$\left|\sum_{i=4}^{\infty} C_i\left(A_2z^2 + A_3z^3 + A_4z^4\right)^i\right|,$$

which is bounded by

$$\epsilon_1 = C_{max}\sum_{i=4}^{\infty}\left(2^{-k}\right)^i = C_{max}\frac{2^{-4k}}{1-2^{-k}}, \tag{19}$$

where $C_{max} = \max_{i\geq 4}|C_i|$.

Now, let us expand the expression obtained from (7) after having discarded the terms of rank $\geq 4$. We get:

$$\begin{aligned} f(A) \approx\ & C_0 + C_1A + C_2A_2^2z^4 + 2C_2A_2A_3z^5 \\ &+ \left(2C_2A_2A_4 + C_2A_3^2 + C_3A_2^3\right)z^6 \\ &+ \left(2C_2A_3A_4 + 3C_3A_2^2A_3\right)z^7 \\ &+ \left(C_2A_4^2 + 3C_3A_2^2A_4 + 3C_3A_2A_3^2\right)z^8 \\ &+ \left(6C_3A_2A_3A_4 + C_3A_3^3\right)z^9 \\ &+ \left(3C_3A_2A_4^2 + 3C_3A_3^2A_4\right)z^{10} \\ &+ 3C_3A_3A_4^2z^{11} + C_3A_4^3z^{12}. \end{aligned} \tag{20}$$

In this rather complicated expression, let us discard all the terms of the form $W \times z^j$ such that the maximum possible value of $W$ multiplied by $z^j = 2^{-kj}$ is less than or equal to $z^4$. We then get (8), that is:

$$f(A) \approx C_0 + C_1A + C_2A_2^2z^4 + 2C_2A_2A_3z^5 + C_3A_2^3z^6.$$

To get a bound on the error $\epsilon$ obtained when approximating (20) by (8), we replace the $A_i$s by their maximum value $2^k - 1$ and we replace the $C_i$s by their absolute value. This gives:

$$\begin{aligned} \epsilon_2 \leq\ & (3|C_2| + 3|C_3|)2^{-4k} + (2|C_2| + 6|C_3|)2^{-5k} \\ &+ (|C_2| + 7|C_3|)2^{-6k} \\ &+ 6|C_3|2^{-7k} + 3|C_3|2^{-8k} + |C_3|2^{-9k} \\ \leq\ & \left(3|C_2| + 3|C_3| + 8.5\max\{|C_2|, |C_3|\} \times 2^{-k}\right)2^{-4k}, \end{aligned}$$

assuming that $8 \times 2^{-k} + 6 \times 2^{-2k} + 3 \times 2^{-3k} + 2^{-4k} < 0.5$, which is true for $k \geq 5$.

As explained in Section 2, when computing (8), we will make another approximation: after having computed $A_2^2$, the computation of $A_2^3$ would require a $2k \times k$ multiplication. Instead of this, we will take the most $k$ significant bits of $A_2^2$ only and multiply them by $A_2$. If we write:

$$A_2^2 = \left(A_2^2\right)_{low} + 2^k\left(A_2^2\right)_{high},$$

where $\left(A_2^2\right)_{low}$ and $\left(A_2^2\right)_{high}$ are $k$-bit numbers, the error committed is

$$C_3\left(A_2^2\right)_{low}A_2z^6,$$

whose absolute value is bounded by $\epsilon_3 = |C_3|2^{-4k}$.

By adding the three errors due to the discarded terms, we get the bound given in the theorem.

## Acknowledgments

This research has been partially funded by a joint PICS grant from the French CNRS and MAE, the US National Science Foundation Grant "Effect of Redundancy in Arithmetic Operations on Processor Cycle Time, Architecture, and Implementation." We thank the reviewers for their careful and helpful comments.

## References

[1] M.D. Ercegovac and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations.* Boston: Kluwer Academic, 1994.

[2] P.M. Farmwald, "High Bandwidth Evaluation of Elementary Functions," *Proc. Fifth IEEE Symp. Computer Arithmetic,* K.S. Trivedi and D.E. Atkins, eds., 1981.

[3] H. Hassler and N. Takagi, "Function Evaluation by Table Look-Up and Addition," *Proc. 12th IEEE Symp. Computer Arithmetic,* S. Knowles and W. McAllister, eds., pp. 10-16, July 1995.

[4] C. Iordache and D.W. Matula, "On Infinitely Precise Roundings for Division, Square-Root, Reciprocal and Square-Root Reciprocal," *Proc. 14th IEEE Symp. Computer Arithmetic,* pp. 233-240, 1999.

[5] M. Ito, N. Takagi, and S. Yajima, "Efficient Initial Approximation and Fast Converging Methods for Division and Square Root," *Proc. 12th IEEE Symp. Computer Arithmetic,* S. Knowles and W. McAllister, eds., pp. 2-9, July 1995.

[6] V.K. Jain and L. Lin, "High-Speed Double Precision Computation of Nonlinear Functions," *Proc. 12th IEEE Symp. Computer Arithmetic,* S. Knowles and W. McAllister, eds., pp. 107-114, July 1995.

[7] V. Lefèvre, J.M. Muller, and A. Tisserand, "Towards Correctly Rounded Transcendentals," *IEEE Trans. Computers,* vol. 47, no. 11, pp. 1,235-1,243, Nov. 1998.

[8] J.M. Muller, *Elementary Functions, Algorithms and Implementation.* Boston: Birkhauser, 1997.

[9] S.F. Oberman and M.J. Flynn, "Division Algorithms and Implementations," *IEEE Trans. Computers,* vol. 46, no. 8, pp. 833-854, Aug. 1997.

[10] D.D. Sarma and D.W. Matula, "Faithful Bipartite ROM Reciprocal Tables," *Proc. 12th IEEE Symp. Computer Arithmetic,* S. Knowles and W. McAllister, eds., pp. 17-28, July 1995.

[11] M. Schulte and J. Stine, "Symmetric Bipartite Tables for Accurate Function Approximation," *Proc. 13th IEEE Symp. Computer Arithmetic,* T. Lang, J.M. Muller, and N. Takagi, eds., 1997.

[12] P.T.P. Tang, "Table Lookup Algorithms for Elementary Functions and Their Error Analysis," *Proc. 10th IEEE Symp. Computer Arithmetic,* P. Kornerup and D.W. Matula, eds., pp. 232-236, June 1991.

[13] W.F. Wong and E. Goto, "Fast Hardware-Based Algorithms for Elementary Function Computations Using Rectangular Multipliers," *IEEE Trans. Computers,* vol. 43, pp. 278-294, Mar. 1994.

[14] W.F. Wong and E. Goto, "Fast Evaluation of the Elementary Functions in Single Precision," *IEEE Trans. Computers,* vol. 44, no. 3, pp. 453-457, Mar. 1995.

[15] T. Lang and P. Montuschi, "Very High Radix Square Root with Prescaling and Rounding and a Combined Division/Square Root Unit," *IEEE Trans. Computers,* vol. 48, no. 8, pp. 827-841, Aug. 1999.

[16] E. Antelo, T. Lang, and J.D. Bruguera, "Computation of $\sqrt{x/d}$ in a Very High Radix Combined Division/Square-Root Unit with Scaling," *IEEE Trans. Computers,* vol. 47, no. 2, pp. 152-161, Feb. 1998.

**Milos D. Ercegovac** earned his BS in electrical engineering (1965) from the University of Belgrade, Yugoslavia, and his MS (1972) and PhD (1975) in computer science from the University of Illinois, Urbana-Champaign. He is a professor in the Computer Science Department, School of Engineering and Applied Science at the University of California, Los Angeles. Dr. Ercegovac specializes in research and teaching in digital arithmetic, digital design, and computer system architecture. His recent research is in the areas of arithmetic design for field programmable gate arrays (FPGAs). His research contributions have been extensively published in journals and conference proceedings. He is a coauthor of two textbooks on digital design and of a monograph in the area of digital arithmetic. Dr. Ercegovac has been involved in organizing the IEEE Symposia on Computer Arithmetic. He served as an editor of the *IEEE Transactions on Computers* and as a subject area editor for the *Journal of Parallel and Distributed Computing*. He is a member of the ACM and the IEEE Computer Society.

**Tomás Lang** received an electrical engineering degree from the Universidad de Chile in 1965, an MS from the University of California (Berkeley) in 1966, and the PhD from Stanford University in 1974. He is a professor in the Department of Electrical and Computer Engineering at the University of California, Irvine. Previously, he was a professor in the Computer Architecture Department of the Polytechnic University of Catalonia, Spain, and a faculty member of the Computer Science Department at the University of California, Los Angeles.

Dr. Lang's primary research and teaching interests are in digital design and computer architecture with current emphasis on high-speed and low-power numerical processors and multiprocessors. He is coauthor of two textbooks on digital systems, two research monographs, one IEEE Tutorial, and author or coauthor of research contributions to scholarly publications and technical conferences. He is a member of the IEEE Computer Society.

**Jean-Michel Muller** received the Engineer degree in applied mathematics and computer science in 1983 and the PhD in computer science in 1985, both from the Institut National Polytechnique de Grenoble, France. In 1986, he joined the CNRS (French National Center for Scientific Research). He is with LIP Laboratory, Ecole Normale Supérieure de Lyon, where he manages the CNRS/ENSL/INRIA Arenaire project. His research interests are in computer arithmetic. Dr. Muller served as co-program chair of the 13th IEEE Symposium on Computer Arithmetic and general chair of the 14th IEEE Symposium on Computer Arithmetic. He has been an associate editor of the *IEEE Transactions on Computers* since 1996. He is a member of the IEEE Computer Society.

**Arnaud Tisserand** received the MSc degree and the PhD degree in computer science from the Ecole Normale Supérieure de Lyon, France, in 1994 and 1997, respectively. In 1999, he joined INRIA (National Institute for Computer Science and Control, France), posted to the CNRS/ENSL/INRIA Arenaire project, at the Ecole Normale Supérieure de Lyon. His research interests include computer arithmetic, computer architecture, and VLSI design. He is a member of the IEEE.